APPENDIX A

The following is an alternative implementation of the present invention. The following code is neither annotated nor further described, but is readily understood by ordinarily skilled software developers, especially given the thorough description the invention in previous subsections. The following code contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

```
#define UPDATE_CENTROID
#define USE_MORPH_CLOSE
#define USE_SCALAR_KMEANS_INITIALIZATION
#define USE_LPF_IN_KMEANS

#define MY_NEG_INF      -HUGE_VAL/100
#define MY_POS_INF      HUGE_VAL/100
#define ALAW_COMPAND_PARAM          25          // parameter A for Alaw
Companding .
#define MAX_INTENSITY_VALUE         65535   // input signal peak magnitude

inline double
log_gaussian_prob(double x, double mean, double istd)
{ //un-normalized and unscaled by std. dev. log of gaussian prob
      if (mean==0) return MY_NEG_INF;
      else if (istd==0) return 0.0;
      else {
            double z=(x-mean)*istd;
            return (-z*z);
      }
}

inline double
pos_prob(int type, double r, double nr)
{ //un-normalized position prob for FG (type=1) or BG (type=0)
      const double RFACTOR=0.5;
      static double nr_LO_NOMR, nr_HI_NOMR, nr_DI_NOMR;

      switch (type) {
      case 0:
            if (r<=nr_LO_NOMR) return 0.0;
            else if (r<=nr_HI_NOMR) return (r-nr_LO_NOMR)*(nr_DI_NOMR);
            else return 1.0;
      case 1:
            if (r<=nr_LO_NOMR) return 1.0;
            else if (r<=nr_HI_NOMR) return (nr_HI_NOMR-r)*(nr_DI_NOMR);
            else return 0.0;
      case 2:
            nr_LO_NOMR=nr*(1.0-RFACTOR);
            nr_HI_NOMR=nr*(1.0+RFACTOR);
            nr_DI_NOMR=1.0/(nr_HI_NOMR-nr_LO_NOMR);
            return 0.0;
      }
      return 0.0;
```

```
}

inline double
pos_probCu(int type, double r, double nr)
{ //un-normalized position prob for FG (type=1) or BG (type=0)
      const double RFACTOR=0.5;
      static double nr_LO_NOMR, nr_HI_NOMR, mult_factor;

      switch (type) {
      case 0:
            if (r<=nr_LO_NOMR) return 0.0;
            else if (r<=nr_HI_NOMR) {
                  double t=(r-nr);
                  return 0.5-mult_factor*t*t*t;
            }
            else return 1.0;
      case 1:
            if (r<=nr_LO_NOMR) return 1.0;
            else if (r<=nr_HI_NOMR) {
                  double t=(r-nr);
                  return 0.5+mult_factor*t*t*t;
            }
            else return 0.0;
      case 2:
            mult_factor=(nr*RFACTOR);
            mult_factor=0.5/(mult_factor*mult_factor*mult_factor);
            nr_LO_NOMR=nr*(1.0-RFACTOR);
            nr_HI_NOMR=nr*(1.0+RFACTOR);
            return 0.0;
      }
      return 0.0;
}

void
GetInitialClassificationNomRad(long lXStart, long lXStop, long lYStart,
long lYStop,
                  double dCentroid_x, double dCentroid_y, double nom_rad,
CTArray<bool> &pBin)
{
      long p,x,y;
      double r;
      for (p=0,y=lYStart;y<=lYStop;y++)
            for (x=lXStart;x<=lXStop;x++,p++) {
                  r=sqrt((x-dCentroid_x)*(x-dCentroid_x)+(y-
dCentroid_y)*(y-dCentroid_y));
                  pBin[p]=(r<=nom_rad);
            }
}

void
DoHoleFill(long lXStart, long lXStop, long lYStart, long lYStop,
CTArray<bool> &pBin)
{
      long lRowSize=lXStop-lXStart+1;
      long lNumPixels=pBin.GetSize();
      long p,x,y;
#ifdef USE_MORPH_CLOSE
      // Morphological Close operation in 3x3 cross window
      CTArray<bool> tBin;
      tBin.SetSize(lNumPixels);
      // Copy edge pixels
      for (p=0;p<lRowSize;p++) tBin[p]=pBin[p];
```

```
        for (y=lYStart+1;y<lYStop;y++,p+=lRowSize) {
              tBin[p]=pBin[p]; tBin[p+lRowSize-1]=pBin[p+lRowSize-1];
        }
        for (;p<lNumPixels;p++) tBin[p]=pBin[p];
        // Dilate
        for (p=lRowSize+1,y=lYStart+1;y<lYStop;y++,p+=2)
              for (x=lXStart+1;x<lXStop;x++,p++)
                    tBin[p] = (pBin[p] || pBin[p-1] || pBin[p+1] || pBin[p-
lRowSize] || pBin[p+lRowSize]);
        // Erode
        for (p=lRowSize+1,y=lYStart+1;y<lYStop;y++,p+=2)
              for (x=lXStart+1;x<lXStop;x++,p++)
                    pBin[p] = (tBin[p] && tBin[p-1] && tBin[p+1] && tBin[p-
lRowSize] && tBin[p+lRowSize]);
#else
        // A simple filling operation to fill stray BG pixels surrounded by
        // 4-connected FG pixels and vice-versa
        for (p=lRowSize+1,y=lYStart+1;y<lYStop;y++,p+=2) {
              for (x=lXStart+1;x<lXStop;x++,p++) {
                    if (pBin[p]==false)
                          pBin[p] = (pBin[p-1] && pBin[p+1] && pBin[p-
lRowSize] && pBin[p+lRowSize]);
                    else
                          pBin[p] = (pBin[p-1] || pBin[p+1] || pBin[p-
lRowSize] || pBin[p+lRowSize]);
              }
        }
#endif
        return;
}

// param provides the mu or A value for companding. InPk provides the input
signal peak magnitude.
bool
ALawCompand(CTArray< CTArray<unsigned short> > &ppPixelData, int param,
unsigned short InPk)
{
        double lnAp1, InPkdA;
        long  p, c;
//      CTArray< CTArray<unsigned short> > ppPixelDataALaw;
        long lNumColors=ppPixelData.GetSize();
        long lNumPixels=ppPixelData[0].GetSize();
        lnAp1 = log(param) + 1;
      InPkdA    = InPk / param;
      // A-law compressor
        for (p=0;p<lNumPixels;p++)
              for (c=0;c<lNumColors;c++)
              {
              if((ppPixelData[c][p]) <= InPkdA )
                          ppPixelData[c][p]= (unsigned short)(param / lnAp1 *
ppPixelData[c][p]+0.5);
              else if((ppPixelData[c][p]) >  InPkdA);
                          ppPixelData[c][p]= (unsigned short)(InPk / lnAp1 *
(1 + ppPixelData[c][p]) / InPkdA + 0.5);
              }
              return 1;
}
bool
GetInitialClassificationScalarKMeans(long lXStart, long lXStop, long
lYStart, long lYStop,
                                                                  CTArray<
CTArray<unsigned short> > &ppPixelData, CTArray<bool> &pBin,
```

```
                                                                  int
num_noise_thresh)
{
      const long MAX_KM_ITER=32;
      const long MAX_NO_TRY=3;
      const double CONVERGENCE_FACTOR=0.001;
      const long weakthresh=20;

      long lNumColors=ppPixelData.GetSize();
      long lNumPixels=ppPixelData[0].GetSize();
      unsigned long l[2], lsum[2], lt, ltprev;
      long p,c,n[2],m[2],lNumIter, lNumTry;
      bool bIsConverged,bIsDone,ex;
      CTArray<unsigned long> pPixelSumData;
      CTArray<bool> mBin;

      pPixelSumData.SetSize(lNumPixels);
      mBin.SetSize(lNumPixels);

      //Computes sum of pixels for 2-means
      for (p=0;p<lNumPixels;p++)
            for (pPixelSumData[p]=0,c=0;c<lNumColors;c++)
                  pPixelSumData[p]+=((unsigned long)ppPixelData[c][p]);

#ifdef USE_LPF_IN_KMEANS
      //Low pass filter data
      CTArray<unsigned long> pPixelSumFiltData;
      pPixelSumFiltData.SetSize(lNumPixels);
      long lRowSize=lXStop-lXStart+1,x,y;
      for (p=0;p<lRowSize;p++) pPixelSumFiltData[p]=pPixelSumData[p];
      for (y=lYStart+1;y<lYStop;y++,p+=lRowSize) {
            pPixelSumFiltData[p]=pPixelSumData[p];
pPixelSumFiltData[p+lRowSize-1]=pPixelSumData[p+lRowSize-1];
      }
      for (;p<lNumPixels;p++) pPixelSumFiltData[p]=pPixelSumData[p];
      for (p=lRowSize+1,y=lYStart+1;y<lYStop;y++,p+=2) {
            for (x=lXStart+1;x<lXStop;x++,p++) {

      pPixelSumFiltData[p]=(((pPixelSumData[p]<<2)+pPixelSumData[p+1]+pPixe
lSumData[p-1]+

pPixelSumData[p+lRowSize]+pPixelSumData[p-lRowSize]+4)>>3);
            }
      }
      for (p=0;p<lNumPixels;p++) pPixelSumData[p]=(pPixelSumFiltData[p]);
#endif

      for (p=0;p<lNumPixels;p++) mBin[p]=1;

      bIsDone=false; lNumTry=0;
      while (!bIsDone ) {
            //Computes max and min of sum
            l[0]=(65535*lNumColors);l[1]=(0);
            for (p=0;p<lNumPixels;p++) {
                  if (mBin[p]) {
                        l[1]=(pPixelSumData[p]>l[1]?pPixelSumData[p]:l[1]);
                        l[0]=(pPixelSumData[p]<l[0]?pPixelSumData[p]:l[0]);
                  }
            }
            if (l[0]==l[1]) return 0;
            //Do k-means for 2 classes
            lt=(l[0]+l[1]);
```

```
                bIsConverged=false;lNumIter=0;
                while (!bIsConverged && lNumIter<MAX_KM_ITER) {
                        ltprev=lt;
                        lsum[0]=lsum[1]=0.0;
                        n[0]=n[1]=0;
                        m[0]=m[1]=0;
                        for (p=0;p<lNumPixels;p++) {
                                pBin[p]=((pPixelSumData[p]<<1)>lt);
                                n[pBin[p]]++;
                                if (mBin[p]) {
                                        m[pBin[p]]++;
                                        lsum[pBin[p]]+=(double)pPixelSumData[p];
                                }
                        }
                        if (m[0]) l[0]=((double)lsum[0]/m[0]+0.5);
                        if (m[1]) l[1]=((double)lsum[1]/m[1]+0.5);
                        lt=(l[0]+l[1]);
                        bIsConverged=(fabs((double)lt/ltprev-
1)<CONVERGENCE_FACTOR);
                        lNumIter++;
                }
                if (bIsConverged==0)
                        printf("Debug\n");
                if (n[1]<num_noise_thresh || n[0]<num_noise_thresh) {
                        ex=(n[1]<num_noise_thresh);
                        for (p=0;p<lNumPixels;p++)
mBin[p]=(mBin[p]?(pBin[p]^ex):0);
                }
                else
                        bIsDone=true;
                lNumTry++;
        }

        return (l[1]-l[0]>(weakthresh*lNumColors));
}

long
ComputeMask(CTArray< CTArray<unsigned short> > &ppPixelData,
                        long lXStart, long lXStop, long lYStart, long lYStop,
double nom_rad,
                        double &dCentroid_x, double &dCentroid_y, bool
&bIsSpotShifted)
{
        const long MAX_EM_ITER=5;
        const long REJECT_ITER=3;
        const double REJECT_Z=3.0;
        const double REJECT_Z_STEP=2.0;
        const double MAX_REJECT_RATIO=0.75;
        const double KMEANS_RAD_REJECT_FACTOR=0.6;
        const double KMEANS_CEN_REJECT_FACTOR=1.5;
        const double KMEANS_MOI_REJECT_FACTOR=2.0;
        const double ISSPOT_FACTOR=0.75;
        const double ISWEAKSPOT_FACTOR=3.0;
        const double GRID_MOVED_REJECT_FACTOR=1.0;

        static double zero=0;

        long p, x, y, c, lNumIter=0,n[2],nR[2];
        bool bIsConverged=false;
        long lNumPixels=(lXStop-lXStart+1)*(lYStop-lYStart+1);
        long lNumColors=ppPixelData.GetSize();
        double dRadius;
```

```
        double gridCentroid_x, gridCentroid_y;


        gridCentroid_x=dCentroid_x;
        gridCentroid_y=dCentroid_y;


        CTArray<bool> pBin, pBinPrev;
        CTArray<double> pPosProb; //Prob(Fg/r)
        CTArray< CTArray<double> > dAve, dStdDev, dSum, dSumSq, dSumR,
dSumSqR;

        pBin.SetSize(lNumPixels);
        pBinPrev.SetSize(lNumPixels);

        pPosProb.SetSize(lNumPixels);

        dAve.SetSize(2);
        dAve[0].SetSize(lNumColors);
        dAve[1].SetSize(lNumColors);
        dStdDev.SetSize(2);
        dStdDev[0].SetSize(lNumColors);
        dStdDev[1].SetSize(lNumColors);
        dSum.SetSize(2);
        dSum[0].SetSize(lNumColors);
        dSum[1].SetSize(lNumColors);
        dSumSq.SetSize(2);
        dSumSq[0].SetSize(lNumColors);
        dSumSq[1].SetSize(lNumColors);
        dSumR.SetSize(2);
        dSumR[0].SetSize(lNumColors);
        dSumR[1].SetSize(lNumColors);
        dSumSqR.SetSize(2);
        dSumSqR[0].SetSize(lNumColors);
        dSumSqR[1].SetSize(lNumColors);

        ALawCompand(ppPixelData, ALAW_COMPAND_PARAM, 65535);

#ifdef USE_SCALAR_KMEANS_INITIALIZATION
        int min1=(int)(3.14159*pow((1.0-
KMEANS_RAD_REJECT_FACTOR)*nom_rad,2.0)+0.5);
        int min0=lNumPixels-
(int)(3.14159*pow((1.0+KMEANS_RAD_REJECT_FACTOR/2)*nom_rad,2.0)+0.5);
        min0=(min0<0?0:min0);
        if (!GetInitialClassificationScalarKMeans(lXStart, lXStop, lYStart,
lYStop, ppPixelData, pBin, (min1<min0?min1:min0)))
        {
              return 0;
        }
        DoHoleFill(lXStart, lXStop, lYStart, lYStop, pBin);
        for (n[0]=n[1]=0,p=0;p<lNumPixels;p++) n[pBin[p]]++;
        dRadius=sqrt((double)n[1]/3.14159);
        bool acceptKM;
        if(dRadius>(1.0-KMEANS_RAD_REJECT_FACTOR)*nom_rad &&
dRadius<(1.0+KMEANS_RAD_REJECT_FACTOR)*nom_rad)
              acceptKM=1;
        else
              {
                    return 0;
              }
```

```
        double dtCentroid_x, dtCentroid_y;
        if (acceptKM) {
              //Compute centroid of fg region
              long sum_x=0, sum_y=0;
              double sumsq_r=0.0;
              for (p=0,y=lYStart;y<=lYStop;y++) {
                    for (x=lXStart;x<=lXStop;x++,p++) {
                          if (pBin[p]) {
                                sum_x+=x; sum_y+=y;
                          }
                    }
              }
              if (n[1]) {
                    dtCentroid_x = ((double)sum_x/n[1]);
                    dtCentroid_y = ((double)sum_y/n[1]);
                    if (sqrt((dtCentroid_x-dCentroid_x)*(dtCentroid_x-
dCentroid_x)+(dtCentroid_y-dCentroid_y)*(dtCentroid_y-dCentroid_y))>
KMEANS_CEN_REJECT_FACTOR*nom_rad)
                    {
                          return 0;
                          bIsSpotShifted=1;
                    }

                    sumsq_r=0;
                    for (p=0,y=lYStart;y<=lYStop;y++) {
                          for (x=lXStart;x<=lXStop;x++,p++) {
                                if (pBin[p]) sumsq_r+=(x-dtCentroid_x)*(x-
dtCentroid_x)+(y-dtCentroid_y)*(y-dtCentroid_y);
                          }
                    }
                    if(sumsq_r*3.14159> KMEANS_MOI_REJECT_FACTOR*n[1]*n[1])
                    {
                          return 0;
                          bIsSpotShifted=1;
                    }
                    else acceptKM =1;
              }

              //Accept nom_rad and centroid based on K-Means
              nom_rad=dRadius;
              dCentroid_x=dtCentroid_x;
              dCentroid_y=dtCentroid_y;
        }
        else
#endif
        GetInitialClassificationNomRad(lXStart, lXStop, lYStart, lYStop,
                                                      dCentroid_x,
dCentroid_y, nom_rad, pBin);
        pos_prob(2,0,nom_rad);
        //Computes radii and normalization factors
              for (p=0,y=lYStart;y<=lYStop;y++)
                    for (x=lXStart;x<=lXStop;x++,p++) {
                          dRadius=sqrt((x-dCentroid_x)*(x-dCentroid_x)+(y-
dCentroid_y)*(y-dCentroid_y));
                          pPosProb[p]=pos_prob(1,dRadius,nom_rad);
              }


        //EM iterations start
        while (!bIsConverged && lNumIter<MAX_EM_ITER ) {
```

```
            double metric[2], precomp_ratio, reject_z[2];
            double t;

            //Computes averages and std. devs. of two classes

            dSum[0].Initialize(zero);
            dSum[1].Initialize(zero);
            dSumSq[0].Initialize(zero);
            dSumSq[1].Initialize(zero);
            for (c=0;c<lNumColors;c++) {
                  for (n[0]=n[1]=p=0;p<lNumPixels;p++) {
                        n[pBin[p]]++;
                        dSum[pBin[p]][c]+=(double)ppPixelData[c][p];
      dSumSq[pBin[p]][c]+=(double)ppPixelData[c][p]*(double)ppPixelData[c][
p];
                  }
                  t=dAve[0][c]=(n[0]?dSum[0][c]/n[0]:0);
                  t=dAve[1][c]=(n[1]?dSum[1][c]/n[1]:0);
                  t=dStdDev[0][c]=(n[0]>1?sqrt(dSumSq[0][c]-
dAve[0][c]*dAve[0][c])/(n[0]-1):0);
                  t=dStdDev[1][c]=(n[1]>1?sqrt(dSumSq[1][c]-
dAve[1][c]*dAve[1][c])/(n[1]-1):0);
            }

            //reject outliers and recompute
            for (c=0;c<lNumColors;c++) {
                  reject_z[0]=REJECT_Z*dStdDev[0][c];
                  reject_z[1]=REJECT_Z*dStdDev[1][c];;
                  for (int l=0;l<REJECT_ITER;l++) {
                        dSumR[0][c]=dSum[0][c];
                        dSumR[1][c]=dSum[1][c];
                        dSumSqR[0][c]=dSumSq[0][c];
                        dSumSqR[1][c]=dSumSq[1][c];
                        for (nR[0]=nR[1]=p=0;p<lNumPixels;p++) {
                              if (fabs(((double)ppPixelData[c][p]-
dAve[pBin[p]][c]))>=reject_z[pBin[p]]) {
                                    dSumR[pBin[p]][c]-
=(double)ppPixelData[c][p];
                                    dSumSqR[pBin[p]][c]-
=(double)ppPixelData[c][p]*(double)ppPixelData[c][p];
                              } else
                                    nR[pBin[p]]++;
                        }
                        if (nR[0]>n[0]*MAX_REJECT_RATIO) {

      t=dAve[0][c]=(nR[0]?dSumR[0][c]/nR[0]:dAve[0][c]);
                              t=dStdDev[0][c]=(nR[0]>1?sqrt(dSumSqR[0][c]-
dAve[0][c]*dAve[0][c])/(nR[0]-1):dStdDev[0][c]);
                        }
                        else reject_z[0]+=REJECT_Z_STEP*dStdDev[0][c];

                        if (nR[1]>n[1]*MAX_REJECT_RATIO) {

      t=dAve[1][c]=(nR[1]?dSumR[1][c]/nR[1]:dAve[1][c]);
                              t=dStdDev[1][c]=(nR[1]>1?sqrt(dSumSqR[1][c]-
dAve[1][c]*dAve[1][c])/(nR[1]-1):dStdDev[1][c]);
                        }
                        else reject_z[1]+=REJECT_Z_STEP*dStdDev[1][c];
                  }
            }
```

```
            bool bIsSpot=false;
            bool bIsStrongSpot=false;
            for (c=0;c<lNumColors;c++) {
                  bIsSpot|=(fabs(dAve[1][c]-
dAve[0][c])>(dStdDev[1][c]+dStdDev[0][c])*ISSPOT_FACTOR);
                  bIsStrongSpot|=(fabs(dAve[1][c]-
dAve[0][c])>(dStdDev[1][c]+dStdDev[0][c])*ISWEAKSPOT_FACTOR);
            }
            if(!bIsSpot) return 0;
#ifdef USE_SCALAR_KMEANS_INITIALIZATION
            if (!bIsStrongSpot && (lNumIter>0 || acceptKM)) break;
#else
            if (!bIsStrongSpot && lNumIter>0) break;
#endif

            pBinPrev=pBin;

            //Update precomp_ratio for std. devs
            precomp_ratio=1.0;
            for (c=0;c<lNumColors;c++) {
                  dStdDev[0][c]=(dStdDev[0][c]?1.0/dStdDev[0][c]:0.0);
                  dStdDev[1][c]=(dStdDev[1][c]?1.0/dStdDev[1][c]:0.0);
      precomp_ratio*=(dStdDev[1][c]?dStdDev[1][c]:1.0)/(dStdDev[0][c]?dStdD
ev[0][c]:1.0);
            }
            precomp_ratio=2.0*log(precomp_ratio);

            //Classify pixels
            for (p=0,y=lYStart;y<=lYStop;y++) {
                  for (x=lXStart;x<=lXStop;x++,p++) {
                        metric[1]=pPosProb[p];
                        metric[0]=1.0-metric[1];

                        if (metric[1]==1) pBin[p]=1;
                        else if (metric[1]==0) pBin[p]=0;
                        else {
                              metric[1]=2.0*log(metric[1]/metric[0]);
                              metric[0]=0;
                              for (c=0;c<lNumColors;c++) {
                                    t=(double)ppPixelData[c][p];
      metric[0]+=log_gaussian_prob((double)ppPixelData[c][p],dAve[0][c],dSt
dDev[0][c]);
      metric[1]+=log_gaussian_prob((double)ppPixelData[c][p],dAve[1][c],dSt
dDev[1][c]);
                              }
                              pBin[p]=(metric[1]+precomp_ratio>metric[0]);
                        }
                  }
            }

            //Convergence test
            for (bIsConverged=true,p=0;p<lNumPixels && bIsConverged;p++)
                  bIsConverged=(pBinPrev[p]==pBin[p]);
            lNumIter++;

      }
```

```
        DoHoleFill(lXStart, lXStop, lYStart, lYStop, pBin);

        //Compute number of fg pixels
        for (n[0]=n[1]=0,p=0;p<lNumPixels;p++) n[pBin[p]]++;

        #ifdef UPDATE_CENTROID
              //Compute centroid of fg region
              long sum_x=0, sum_y=0;
              for (p=0,y=lYStart;y<=lYStop;y++) {
                    for (x=lXStart;x<=lXStop;x++,p++) {
                          if (pBin[p]) {
                                sum_x+=x; sum_y+=y;
                          }
                    }
              }
              if (n[1]) {
                    dCentroid_x = ((double)sum_x/n[1]);
                    dCentroid_y = ((double)sum_y/n[1]);
              }
        #endif

        //Final check for no spots
        nom_rad=sqrt((double)n[1]/3.14159);
        GetInitialClassificationNomRad(lXStart, lXStop, lYStart, lYStop,
                                                      dCentroid_x,
    dCentroid_y, nom_rad, pBin);
        dSum[0].Initialize(zero);
        dSum[1].Initialize(zero);
        dSumSq[0].Initialize(zero);
        dSumSq[1].Initialize(zero);
        for (c=0;c<lNumColors;c++) {
              for (n[0]=n[1]=p=0;p<lNumPixels;p++) {
                    n[pBin[p]]++;
                    dSum[pBin[p]][c]+=(double)ppPixelData[c][p];

        dSumSq[pBin[p]][c]+=(double)ppPixelData[c][p]*(double)ppPixelData[c][
    p];
              }
              double t=dAve[0][c]=(n[0]?dSum[0][c]/n[0]:0);
              t=dAve[1][c]=(n[1]?dSum[1][c]/n[1]:0);
              t=dStdDev[0][c]=(n[0]>1?sqrt(dSumSq[0][c]-
    dAve[0][c]*dAve[0][c])/(n[0]-1):0);
              t=dStdDev[1][c]=(n[1]>1?sqrt(dSumSq[1][c]-
    dAve[1][c]*dAve[1][c])/(n[1]-1):0);
        }
        bool bIsSpot=false;
        for (c=0;c<lNumColors;c++) {
              bIsSpot|=(fabs(dAve[1][c]-
    dAve[0][c])>(dStdDev[1][c]+dStdDev[0][c])*ISSPOT_FACTOR);
        }
        if(!bIsSpot)
        {
              return 0;
        }

      bIsSpotShifted=(sqrt((gridCentroid_x-dCentroid_x)*(gridCentroid_x-
    dCentroid_x)+(gridCentroid_y-dCentroid_y)*(gridCentroid_y-
    dCentroid_y))>GRID_MOVED_REJECT_FACTOR*nom_rad);

        return n[1];
              }
```